IEICE TRANS. INF. & SYST., VOL.E99–D, NO.12 DECEMBER 2016

LETTER

Fine-Grained Data Management for DRAM/SSD Hybrid Main Memory Architecture*

Liyu WANG[†a)], ***Member***, **Qiang WANG**[†], **Lan CHEN**[†b)], ***and*** **Xiaoran HAO**[†], ***Nonmembers***

SUMMARY Many data-intensive applications need large memory to boost system performance. The expansion of DRAM is restricted by its high power consumption and price per bit. Flash as an existing technology of Non-Volatile Memory (NVM) can make up for the drawbacks of DRAM. In this paper, we propose a hybrid main memory architecture named SSDRAM that expands RAM with flash-based SSD. SSDRAM implements a runtime library to provide several transparent interfaces for applications. Unlike using SSD as system swap device which manages data at a page level, SSDRAM works at an application object granularity to boost the efficiency of accessing data on SSD. It provides a flexible memory partition and multi-mapping strategy to manage the physical memory by micro-pages. Experimental results with a number of data-intensive workloads show that SSDRAM can provide up to 3.3 times performance improvement over SSD-swap.

key words: *data-intensive application, NVM, SSD, hybrid memory*

1. Introduction

With the development of Big Data, data-intensive applications increase requirements of main memory capacity to avoid the high cost of accessing disk. DRAM has been used as main memory for a long time. However, DRAM has constraints of high power consumption and price per bit. Fortunately, the advent of Non-Volatile Memory (NVM) technologies offers a promising opportunity for memory organization. Some projects propose to construct hybrid main memory using NVM devices such as PCM and STT-RAM [1], [2]. Nevertheless, the above researches are usually proceeded using architecture simulators and empirical models because mature products based on these memory technologies are not yet available. Among various NVM devices, flash memory as the existing technology is well-known to bridge the performance gap between DRAM main memory and hard disk [3], [4]. The energy-efficiency and cost-efficiency of flash memory make it more economical than DRAM. On the other hand, flash memory has some inherent defects such as high access latency and limited endurance, which restrict flash as a substitute of DRAM.

Existing usages of flash memory focus on the following aspects. Kgil et al. [5] proposed FlashCache using flash as a disk cache which can enhance disk performance but not main memory. Some proposals realized hybrid memory with hardware extension [6], [7]. The software approaches using SSD as a main memory extension will be introduced in detail in the next section.

In this paper, we propose a hybrid main memory architecture named SSDRAM using SSD as the extension of main memory and using DRAM as the cache of SSD. SSDRAM provides several interfaces to allocate and free memory for applications. It works at an object granularity to boost the efficiency of accessing data on SSD [8]. To reduce the waste of physical memory caused by object granularity, SSDRAM provides a flexible memory partition strategy and a multi-mapping strategy to manage the physical memory by micro-pages.

2. Related Work

Some studies use SSD as a system swap device instead of HDD [9], [10]. Although this does not need application redesigning, it has been proved that SSD-swap cannot explore the raw performance of flash memory [9], [12]. As system virtual memory works at a page (4KB) level, it leads to swap a full page with flash even if only one byte needs to be accessed. This wastes SSD I/O bandwidth and increases latency.

W. Chao et al. [11] use SSD as a secondary memory partition and develop a NVMalloc library for applications to access a distributed NVM store. The NVM devices are mapped into memory in a byte-addressable fashion using the POSIX mmap() interface. Brian Van et al. [14] redesign the mmap() interface to fulfill an extended main memory system using SSD with thread level concurrency. However, applications using the above systems need to be aware of the data access patterns for hot data caching. What's more, using SSD as memory via mmap() interface can achieve only a fraction of the SSD's performance and incur significant runtime overhead [12].

A. Badam et al. [12] present a hybrid memory named SSDAlloc using SSD as an object store. It exposes flash via page-based virtual memory interface, while internally works at an object granularity. At the virtual memory level, SSDAlloc allocates only one object per page. At the physical memory level, SSDAlloc divides the RAM into two cache layers, a page buffer and an object cache. The page buffer offers direct access for applications but wasting RAM space.

Manuscript received May 13, 2016.
Manuscript revised July 23, 2016.
Manuscript publicized August 30, 2016.

[†]The authors are with the Institute of Microelectronics of Chinese Academy of Sciences, Beijing, China.

*This work was supported in part by the National Science and Technology Major Project of China under grant 2013ZX03001008-003.

a) E-mail: wangliyu@ime.ac.cn
b) E-mail: chenlan@ime.ac.cn

DOI: 10.1587/transinf.2016EDL8105

The object cache stores objects compactly while cannot be accessed directly by applications. So extra page faults, data mapping and copying are caused even when objects are located in the object cache of RAM.

Unlike SSDAlloc, Chameleon [13] provides direct access to the entire physical memory for applications without extra page faults. It provides object-based virtual memory with multiple virtual page sizes ranging from 512 bytes to 4KB that is power of 2. However, objects with sizes out of these fixed values cause memory fragmentation that wastes RAM space. For example, a 513B object needs a 1024B physical size. Additionally, the experiments of this work do not include evaluation on varied-sized objects and even multithreaded applications.

3. Design

3.1 The SSDRAM Architecture

SSDRAM implements a library to offer interfaces for applications to allocate and free memory transparently like system standard interfaces *malloc* and *free*. In comparison with using SSD as system swap which manages data at a page level, SSDRAM works at an object granularity to boost system performance and the efficient use of SSD. In order to perform the fine-grained data accessing, a custom page fault handler is realized to intercept system SIGSEGV signals and bypass OS page fault mechanism. To reduce the waste of physical memory caused by object granularity, SSDRAM partitions physical pages into a few micro-pages according to a fixed growth in size. And it implements a multi-mapping strategy which can map multiple virtual pages to the micro-pages of the same physical page.

3.2 Memory Partition and Multi-Mapping

SSDRAM divides physical memory into multiple slabs which include a few pages. And each page is partitioned into some micro-pages to load objects. The minimum size of micro-page is application configurable, then other larger sizes ascend at the step of this minimum size. Breaking through the original OS page boundary limit, micro-pages in various 4KB pages with the same size and same offset are classified as one pattern. A physical page should be assigned to micro-pages with the same size as possible as it can. If the size cannot meet the boundary of a 4KB page, the rest space of this page will be collected to other patterns by picking out larger size from the end of the page. For example, assuming that the minimum micro-page is 256B, the page with partition size 1536B remains 1024B of space. The rest space will be preferred to incorporate into the pattern whose size is 1024B and offset is 3072B instead of four patterns with smaller size 256B and different offset values.

SSDRAM allocates one object per virtual page to implement fine granularity management. Therefore, a multi-mapping strategy is needed to map between multiple virtual pages and micro-pages within one physical page. This requires each object in virtual page to align with the offset of each mapped micro-page.

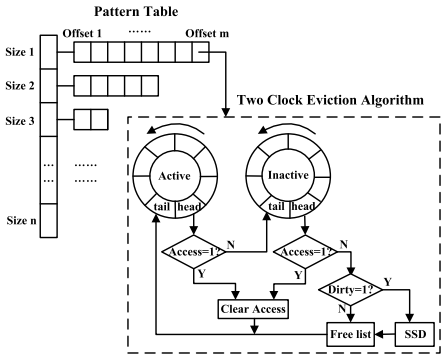


Fig. 1 The management of micro-pages for one pattern.

3.3 Micro-Page Management

SSDRAM keeps a set of lists for each pattern to manage free and used micro-pages. Furthermore, the used micro-pages are organized into an active list and an inactive list. The active list contains micro-pages which have been accessed recently and the inactive list contains micro-pages which have not been used recently. When there is no free micro-page for the required object, SSDRAM adopts a two-fifo clock algorithm for the specific pattern to perform the eviction process as can be seen in Fig. 1, which begins from the head of the inactive list. If the accessed bit in PTE of the object is set, SSDRAM moves the object to the tail of the active list with accessed bit unset. Otherwise, the object is chosen as a candidate for eviction. If it is dirty, it should be written to SSD before reclaiming its micro-page to the free list. When the inactive list is not enough, SSDRAM checks the accessed bit from the head of the active list. If the accessed bit is set, the object will be linked to the tail of the active list and the accessed bit will be unset. If not, the object will be moved to the tail of the inactive list.

3.4 SSD Management

SSD is partitioned into 256KB logical blocks and managed in a log-structured manner. Each block contains several pages and each page owns a few sectors whose size is equal to the minimum size of micro-pages. To minimize the latency of SSD random write operations, SSDRAM assembles the evicted objects into blocks in the write buffer and writes a whole block to SSD every time in the background. SSDRAM creates a mapping table called Sector Table to record the SSD location, object size and offset. Each entry is indexed by virtual memory address which is stored at the head of each block.

Because SSD does not support to write back in place, even the same object should be written to a different SSD location which needs to be updated in the Sector Table. To reclaim the former location on SSD, the garbage collection

strategy scans a SSD block which meets the reclaiming condition and picks out valid objects in the background. A valid object is confirmed by comparing the current SSD location with that stored in the Sector Table. If they are the same, the copy of the object is valid and it needs to be written to a new SSD block. If not, it can be discarded directly for garbage collection.

4. Implement

SSDRAM realizes a runtime library to replace the standard library. It provides interfaces for applications including *nv_malloc*, *nv_calloc*, *nv_realloc and nv_free*.

To perform the communication between user space and OS kernel, SSDRAM implements a linux kernel module which is used to read and modify PTE. In order to reduce the overhead of the communication, SSDRAM binds multiple PTE operations together and packages the information into one message to transfer to the kernel space.

SSDRAM provides the support for multithreaded applications to response the concurrent requests. To reduce the conflicts of multiple threads for the shared data, SSDRAM performs locks with different fine granularities for different operations to achieve more parallelism.

5. Performance Evaluation

In this section, we compare the performance of SSDRAM with SSD-swap using various workloads such as microbenchmarks, Sorting, B-tree and so on.

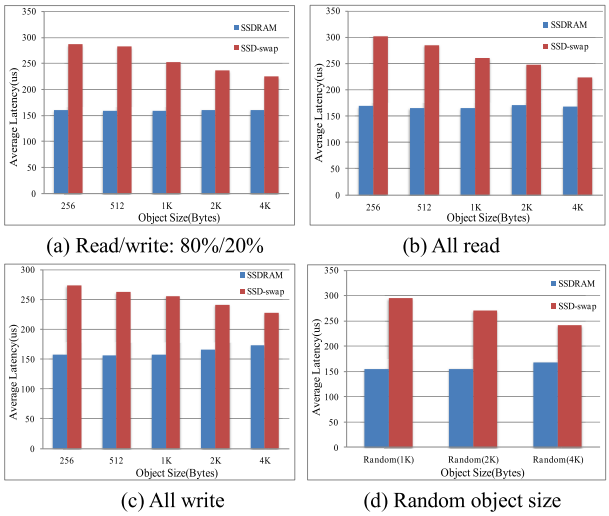
5.1 Experiment Setup

We conduct experiments on a server with a 64-bit Linux 2.6.32 kernel. It owns eight Intel7-4790 3.6GHz CPU and 6GB DRAM. A 120GB Kingston SSD is adopted for SSDRAM and SSD-swap respectively. If main memory is sufficient or object size is larger than 4KB, there is no need to call SSDRAM because it is the same with the standard library. We only evaluate the performance that the capacity of memory is not enough and object size is smaller than 4KB. To simulate the out-of-DRAM behavior, we restrict memory capacity to 64MB for both systems.

5.2 Microbenchmarks

We evaluate the basic performance of random access with certain object size ranging from 256B to 4KB and random object size. In all experiments, 10GB worth of objects are allocated initially, and then 1GB worth of them are accessed randomly.

(1) Read/write ratio of 80%/20%: Figure 2 (a) shows the average latency when random read/write ratio is 80%/20%. It can be seen that SSDRAM outperforms SSD-swap obviously, especially with the decrease of object size. For 512B object size, SSDRAM obtains an object in 158 μs, while SSD-swap needs 282 μs, which incurs 78.5%



(a) Read/write: 80%/20% (b) All read

(c) All write (d) Random object size

Fig. 2 The average latency of random access under microbenchmarks.

higher latency. SSD-swap manages data at a page granularity, which leads to a waste of IO bandwidth. The smaller the object size is, the less valid data each physical page can fetch from system swap. SSDRAM operates data at an object granularity which can improve the utilization of IO bandwidth.

(2) All read: Figure 2 (b) gives the average latency of random read operations. SSDRAM saves 33.3%-78.7% latency compared to SSD-swap for object size ranging from 256 bytes to 4KB. When a required object is not in memory, SSD-swap has to read a full page from SSD and copy excessive data to the memory. Whereas, SSDRAM loads the desired object that is really needed. It can hold more valid and hot objects in memory, which help to improve the hit ratio of application requests and reduce the read operations to SSD.

(3) All write: Figure 2 (c) presents the comparison between both systems of random write operations. SSDRAM performs better than SSD-swap in accordance with the above results. Overwriting an object which is not in memory needs to load data from SSD into memory first. SSDRAM loads the exact data at a finer granularity than SSD-swap. And when there is no space in memory, SSDRAM evicts cold objects instead of full pages. The dirty objects are assembled compactly into blocks in the write buffer and evicted to SSD in the background.

(4) Random object size: Figure 2 (d) shows the average latency of accessing random object size when read/write ratio is 80%/20%. Random (4K) means that object size is distributed randomly from 8 bytes to 4KB. SSDRAM outperforms SSD-swap in three sets of object size under different ranges. For the situation of object size ranging from 8 bytes to 1KB, SSD-swap consumes 90.3% more latency than SSDRAM. Because SSDRAM provides a flexible memory partition strategy, which makes more effective use of physical memory and provides better support for accessing objects with random size than that of SSD-swap as long as the ob-

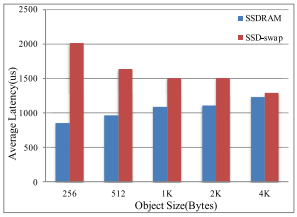


Fig. 3 Sorting workload.

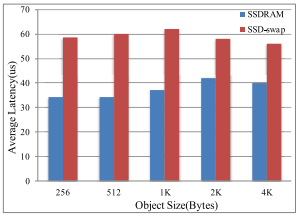


Fig. 5 Bloomfilter workload.

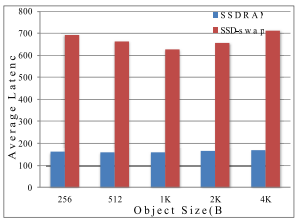


Fig. 4 B-tree workload.

Fig. 6 Multiple threads.

ject size is smaller than 4KB page.

5.3 Sorting Workload

A quick-sort workload is adopted to test random access and the data swap mechanism. We randomly choose 1GB worth of objects from the 10GB data which are widely distributed in SSD. The objects of the group are sorted in place, which needs random read and write operations.

Figure 3 presents the results of the sorting workload at different object sizes. With the decrease of object size, the latency gap between SSDRAM and SSD-swap becomes greater. For 256B object size, SSDRAM obtains 1.4 times performance gain compared to SSD-swap. When object size reaches page size, the average accessing latency of SSDRAM is close to that of SSD-swap. Because the randomly selected objects are spread across SSD, which need to be read into memory to compare, and written in place. SSD-swap manages data at 4KB page granularity, which caches lots of useless objects in memory. By contrast, SSDRAM works at an object granularity, which holds more valid objects in one physical page when object size is smaller than one page.

5.4 B-Tree Workload

B-tree algorithm is usually used to search data when large-scale data sets are stored on the storage. It holds more keys per node to reduce the height of the tree to improve the search efficiency. To explore the performance of such read-heavy applications, we create a B-tree structure to search 1GB worth of objects from entire 10GB data with several fixed object sizes.

Figure 4 gives the average latency of both systems running the search algorithm. SSDRAM runs 2.9-3.3 times faster compared to SSD-swap for object size ranging from 256 bytes to 4KB. When a key in one node which is not in the memory is required, SSD-swap needs to access two full pages to obtain the node metadata and the object data respectively. This leads to load more invalid data and waste space in physical pages. On the contrary, SSDRAM makes smaller granularity read operations and loads the exact data that is really needed.

5.5 Bloomfilter Workload

We adopt a bloomfilter to verify the performance of write-heavy workloads. It is used to retrieve whether an element is in a set by marking the bits. Bloomfilter consists of a long bit vector and a series of random mapping functions. A vector of 80 billion bits (10GB) is created and the bits are managed by objects with certain size. We use 8 bytes keys to index elements and adopt eight various hash functions to map eight positions for each element.

Figure 5 shows the results of running the bloomfilter. The average latency of SSD-swap is much higher than SSDRAM. At 512B object size SSDRAM performs 76.5% better than SSD-swap. Each element has eight positions which may locate in several objects. Therefore, it needs to write more than one object by marking the bits of different locations. SSD-swap produces more dirty physical pages and causes more erasure operations on SSD. In comparison, SSDRAM only evicts dirty objects and amortizes the writing cost by coalescing multiple dirty objects into one block.

5.6 Multiple Threads

At last we give the results of concurrent requests of 1/2/4/8 threads using microbenchmark workload in Fig. 6. SSDRAM needs less runtime than SSD-swap when the number of threads is less than eight. SSDRAM performs different fine grained locks in different phases to achieve more parallelism and fully exploit the SSD's potential. But with the increase of the number of threads, the latency of SSDRAM

has a modest increase which is close to SSD-swap. Therefore, how to improve the ability of supporting more threads is one of our future work.

6. Conclusion

With the increase of the number of data-intensive applications, there is a growing demand to process large-scale data in main memory. Building a large capacity of DRAM memory has become more and more difficult because of the high power consumption and price per bit. Flash-based SSD has some advantages of low power, high density and low price. But it cannot replace DRAM as main memory due to its high latency and limited endurance. Therefore, we construct a hybrid memory SSDRAM which uses SSD as the extension of main memory and DRAM as the cache for SSD. SSDRAM implements a runtime library for applications and works at a fine granularity with a few strategies. The experiment results show that SSDRAM achieves higher performance than SSD-swap under various data-intensive workloads.

As future work, we plan to integrate SSDRAM into more applications to promote performance, such as in-memory databases and so on. Additionally, support for more multithreaded requests is also part of the future work.

References

[1] S. Li, X. Chen, M. Zhou, G. Li, Y. Zhang, and Z. Song, "PCRAM-Aware Cluster Allocation Algorithm for Hybrid Main Memory Hierarchy," IEICE Electron. Express, vol.12, no.5, 2015.

[2] R. Salkhordeh and H. Asadi, "An Operating System Level Data Migration Scheme in Hybrid DRAM-NVM Memory Architecture," Design, Automation & Test in Europe Conference & Exhibition, pp.936–941, 2016.

[3] F. Chen, D.A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," International Conference on Supercomputing, pp.22–32, 2011.

[4] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs," International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, pp.227–236, 2011.

[5] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," Proc. 35th International Symposium on Computer, Architecture, pp.327–338, 2008.

[6] J. Meza, Y. Luo, S. Khan, J. Zhao, and Y. Xie, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," Proc. Workshop on Energy-Efficient Design, 2013.

[7] H. Kawata and S. Oikawa, "A Feasibility Study of Hybrid DRAM and Flash Memory Management Unit," 2014 IIAI 3rd International Conference on Advanced Applied Informatics, pp.694–698, 2014.

[8] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P.B. Gibbons, M.A. Kozuch, T.C. Mowry, and T. Chilimbi, "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," ACM/IEEE 42nd Annual International Symposium on Computer Architecture, pp.79–91, 2015.

[9] M. Saxena and M.M. Swift, "Flashvm: Virtual memory management on flash," Proc. 2010 USENIX Conference on USENIX Annual Technical Conference, pp.14–14, 2010.

[10] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh, "A new linux swap system for flash memory storage devices," Proceedings of the 2008 International Conference on Computational Sciences and Its Applications, pp.151–156, 2008.

[11] C. Wang, S.S. Vazhkudai, X.S. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines," Parallel & Distributed Processing Symposium (IPDPS), pp.957–968, 2012.

[12] A. Badam and V.S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," Proc. 8th USENIX Conference on Networked Systems Design and Implementation, pp.16–16, 2011.

[13] A. Badam, V.S. Pai, and D.W. Nellans, "Better Flash Access via Shape-shifting Virtual Memory Pages," Conference on Timely Results on Operating Systems Principles, pp.1–14, 2013.

[14] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "DI-MMAP – a scalable memory-map runtime for out-of-core data-intensive applications," Cluster Computing, pp.15–28, 2015.